

# Priority Queues and Heapsort

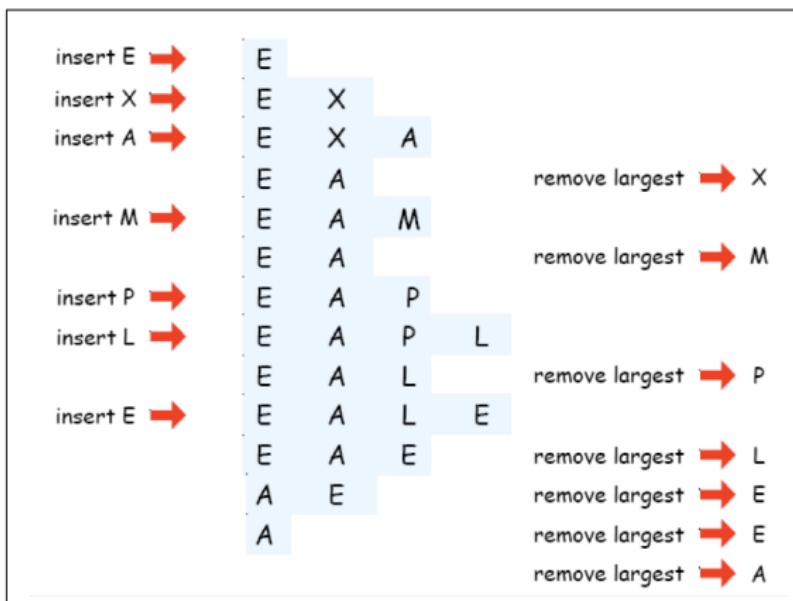
Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

**Acknowledgments:** Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 2.4 and 2.5), Prof. Mhaskar's course slides, and <https://www.cs.princeton.edu/~rs/AlgsDS07/04Sorting.pdf>

# Priority Queue

- A **collection** is a data type that stores a group of items.
- **Priority Queue** is a collection of objects which can be compared. It supports inserting an item, and removing the largest (or smallest) item.



# Priority queue: applications

- Event-driven simulation – customers in a line
- Numerical computation – reducing roundoff error
- Discrete optimization – scheduling
- Operating systems – load balancing, interrupt handling
- Data compression – Huffman codes
- Graph searching – Dijkstra's algorithm, Prim's algorithm
- and many more!

# Priority Queue API

<code>public class MaxPQ&lt;Key extends Comparable&lt;Key&gt;&gt;</code>	
<code>MaxPQ()</code>	<i>create an empty priority queue</i>
<code>MaxPQ(Key[] a)</code>	<i>create a priority queue with given keys</i>
<code>void insert(Key v)</code>	<i>insert a key into the priority queue</i>
<code>Key delMax()</code>	<i>return and remove a largest key</i>
<code>boolean isEmpty()</code>	<i>is the priority queue empty?</i>
<code>Key max()</code>	<i>return a largest key</i>
<code>int size()</code>	<i>number of entries in the priority queue</i>

Key must be Comparable  
(bounded type parameter)

# Priority queue: implementations costs

Challenge: Implement all operations efficiently.

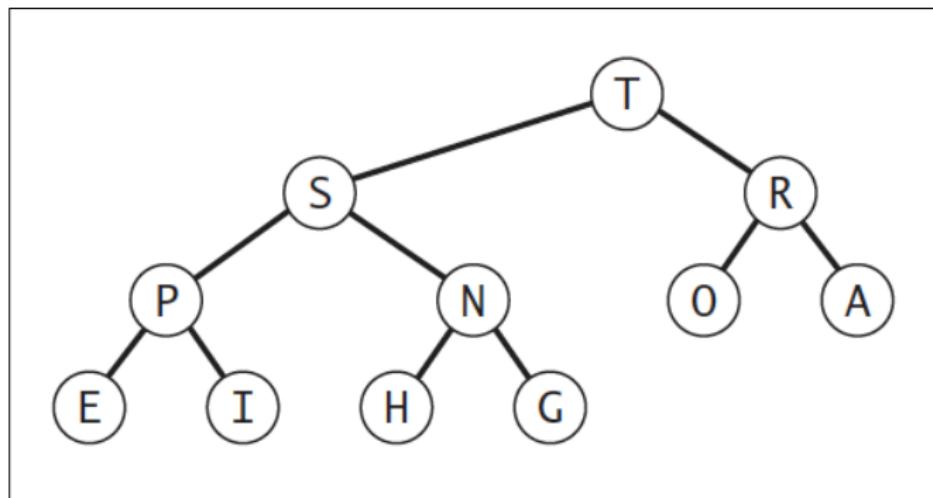
implementation	insert	del max
<b>unordered array</b>	1	$n$
<b>ordered array</b>	$n$	1
<b>goal</b>	$\log n$	$\log n$

Solution: Use a Binary Heap!

# Binary (MAX.) Heap Ordered Tree

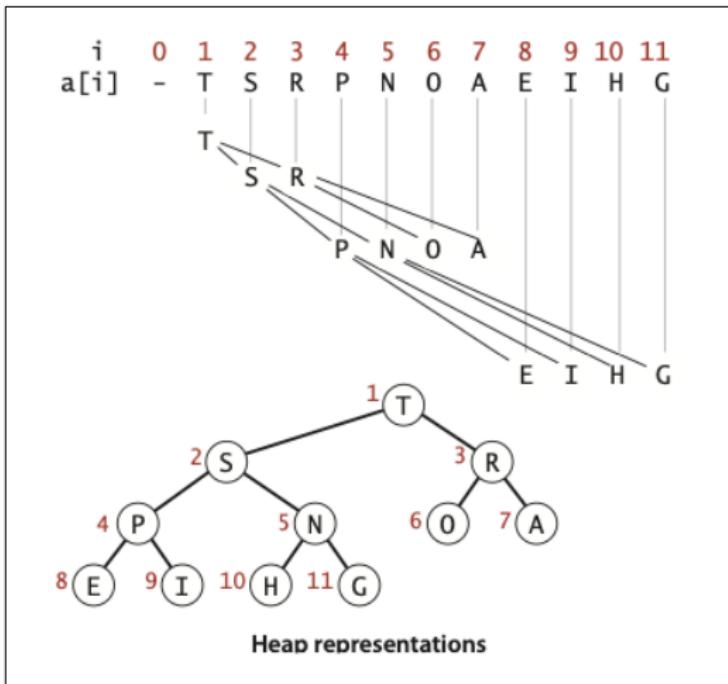
**Binary MAX Heap ordered tree:** is a complete binary tree where

- the keys are in nodes, and
- every parent's key  $\geq$  children's keys (Max. heap property).



## Binary Heap - array representation

- Indices start at 1.
  - Nodes are grouped by level.
  - The root is in position 1.
  - Level 2 uses positions 2 and 3.
  - Level 3 uses positions 4 through 7, etc.
  - The parent of node  $k$  is in position  $\lfloor k/2 \rfloor$ .
  - The two children of  $k$  are in positions  $2k$  and  $2k + 1$ .



# It's Sink or Swim!

Rather than simply sorting an array as quickly as possible, we now want to **maintain the sortedness property**.

- With respect to insertion, we must add new nodes to the end of the heap in order to maintain it as a complete binary tree.
  - We “swim the node up” through the heap until the sortedness property is re-established.
- To remove the maximum node is more complicated than just removing the element at position 1.
  - We swap it with the node at the end and then remove it.
  - Then we have to “sink the node down” (that is, the one we swapped into root position).

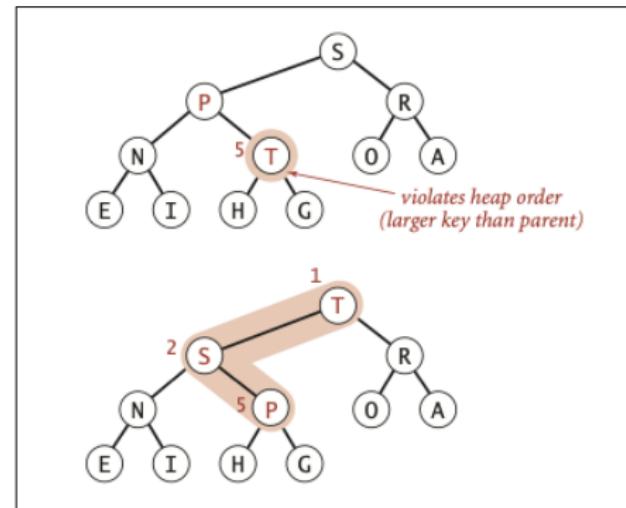
# Swim for it!

If a key is larger than its parent's key it violates the binary heap property. To eliminate the violation:

- Exchange the child's position with its parent.
- Repeat until order is restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

parent of node at k is at k/2

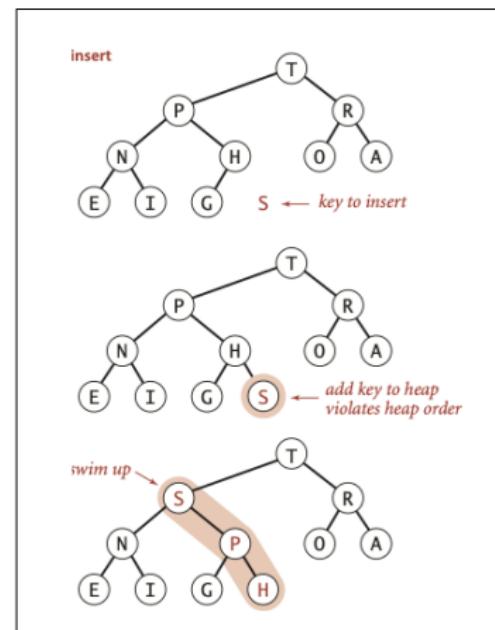


# A Heaping Helping

To insert a node...

- Add node at the minimal unused position in the heap (linearly), then swim it up.
- This will cost at most  $1 + \log_2 n$  comparisons.

```
public void insert(Key x)
{
    pq[++n] = x;
    swim(n);
}
```



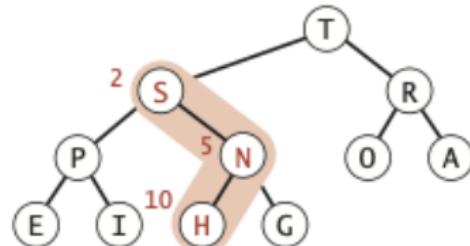
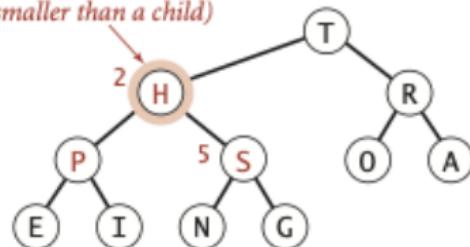
# Binary heap: sink operation

If a key is smaller than any of its children...

- Exchange the parent with the *larger* child.
- Repeat until order is restored.

```
private void sink(int k)
{
    while (2*k <= n)           children of node at k
    {                           are 2*k and 2*k+1
        int j = 2*k;           ↘   ↘
        if (j < n && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

violates heap order  
(smaller than a child)

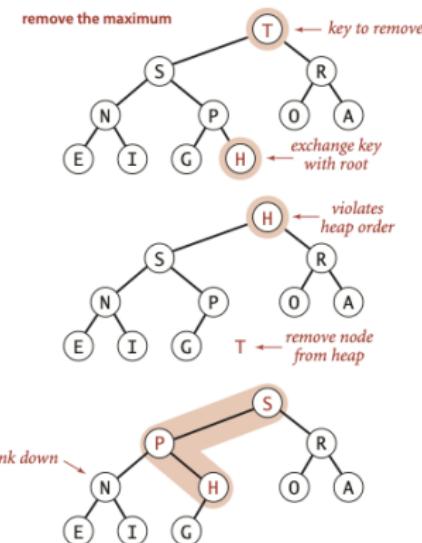


Top-down reheapify (sink)

# Binary heap: delete maximum

- Delete max: Exchange root with node at end, then sink it down.
- Cost: At most  $2 \log_2 n$  comparisons.

```
public Key delMax()
{
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] = null; ← prevent loitering
    return max;
}
```



# Max. Priority Queue

```
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int n;

    public MaxPQ(int capacity)
    {   pq = (Key[]) new Comparable[capacity+1]; }

    public boolean isEmpty()
    {   return n == 0; }

    public void insert(Key key) // see previous code
    public Key delMax() // see previous code

    private void swim(int k) // see previous code
    private void sink(int k) // see previous code

    private boolean less(int i, int j)
    {   return pq[i].compareTo(pq[j]) < 0; }
    private void exch(int i, int j)
    {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t; }

}
```

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

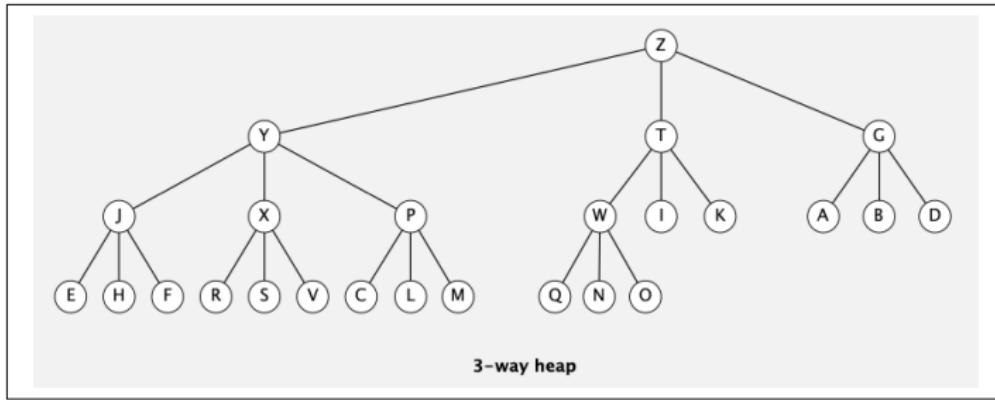
# Priority Queue: implementations cost summary

implementation	insert	del max	max
<b>unordered array</b>	1	$n$	$n$
<b>ordered array</b>	$n$	1	1
<b>binary heap</b>	$\log n$	$\log n$	1

order-of-growth of running time for priority queue with  $n$  items

# Binary heap: practical improvements

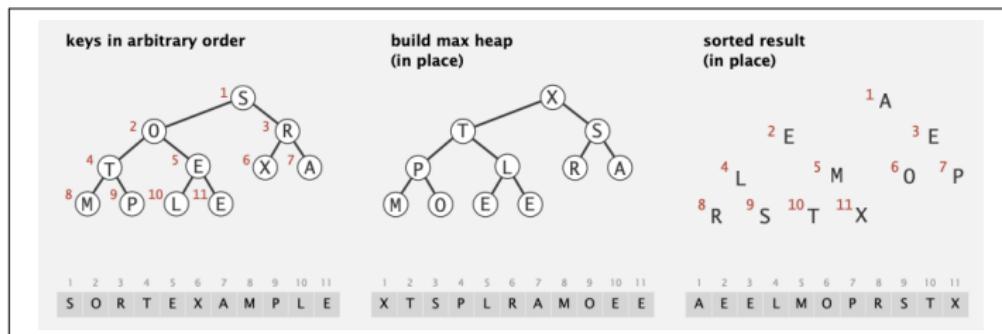
- Increase the number of child nodes per parent node!
- Fun Fact: The height of a complete d-way tree of n nodes is  $\log_d n$ .



# Heapsort Basic Procedure

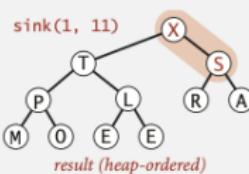
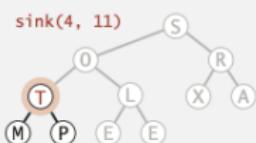
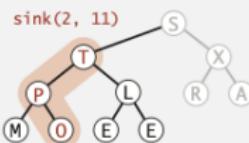
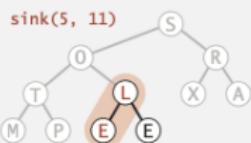
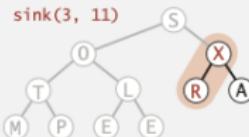
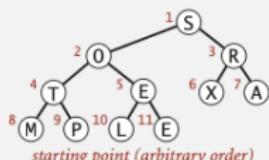
Heapsort is a two-step algorithm

- ➊ Construct a binary heap with the input data.
  - This requires repeated applications of the SINK algorithm until the binary heap property is satisfied.
- ➋ Use the binary heap to construct a sorted array.
  - The root of the binary heap is always maximal, and repeated application of the REMOVEMAX method will *automatically* linearize the heap!



# Heap Construction

```
for (int k = n/2; k >= 1; k--)  
    sink(a, k, n);
```

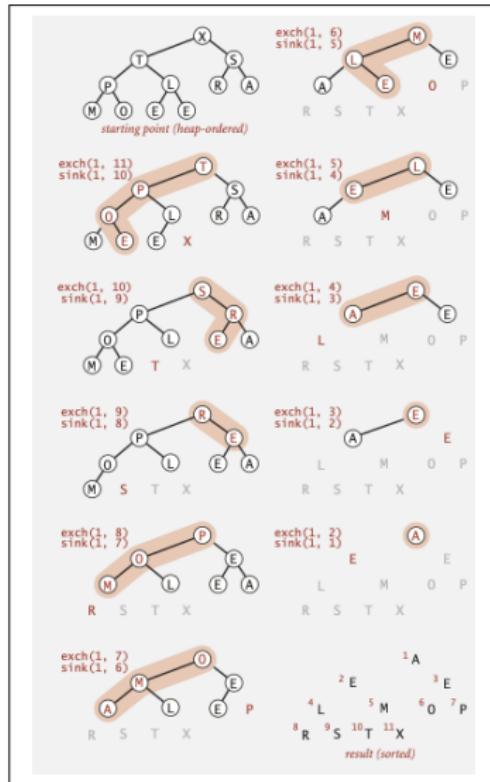


result (heap-ordered)

# Heapsort: Sortdown

- Second pass: Repeatedly remove the maximum.

```
while (n > 1)
{
    exch(a, 1, n--);
    sink(a, 1, n);
}
```



# Heapsort: Java implementation

```
public class Heap
{
    public static void sort(Comparable[] a)
    {
        int n = a.length;
        for (int k = n/2; k >= 1; k--)
            sink(a, k, n);
        while (n > 1)
        {
            exch(a, 1, n);
            sink(a, 1, --n);
        }
    }
    but make static (and pass arguments)
    private static void sink(Comparable[] a, int k, int n)
    { /* as before */ }

    private static boolean less(Comparable[] a, int i, int j)
    { /* as before */ }

    private static void exch(Object[] a, int i, int j)
    { /* as before */ }
}
```

*but convert from 1-based indexing to 0-base indexing*

# Heapsort: Trace

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>		S	O	R	T	E	X	A	M	P	L	E	
11	5	S	O	R	T	L	X	A	M	P	E	E	
11	4	S	O	R	T	L	X	A	M	P	E	E	
11	3	S	O	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
<i>heap-ordered</i>		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R	A	M	E	E	X	
9	1	S	P	R	O	L	E	A	M	E	T	X	
8	1	R	P	E	O	L	E	A	M	S	T	X	
7	1	P	O	E	M	L	E	A	R	S	T	X	
6	1	O	M	E	A	L	E	P	R	S	T	X	
5	1	M	L	E	A	E	O	P	R	S	T	X	
4	1	L	E	E	A	M	O	P	R	S	T	X	
3	1	E	A	E	L	M	O	P	R	S	T	X	
2	1	E	A	E	L	M	O	P	R	S	T	X	
1	1	A	E	E	L	M	O	P	R	S	T	X	
<i>sorted result</i>		A	E	E	L	M	O	P	R	S	T	X	
Heapsort trace (array contents just after each sink)													

# Heapsort: Analysis

**Proposition.** Heap construction uses  $\leq 2n$  compares and  $\leq n$  exchanges.

**Proposition.** Heapsort uses  $\leq 2n \log_2 n$  compares and exchanges. - The algorithm can be improved to  $\approx 1n \log_2 n$ , but no such variant is known to be practical.

**Significance.** In-place sorting algorithm with  $n \log n$  worst-case.

- Mergesort requires extra space. [in-place merge possible, not practical]
- Quicksort requires extra space, worst case is quadratic. [ $n \log n$  worst-case quicksort possible, not practical]
- Heapsort is an improvement in both these areas!

**Bottom line.** Heapsort is optimal for both time and space, but:

- Inner loop longer than quicksort's.
- Makes poor use of cache: array entries are rarely compared with nearby array entries, so the number of cache misses is far higher than for quicksort, mergesort, where most compares are with nearby entries.

# Sorting and Stability

A sorting method is **stable** if it preserves the relative order of equal keys in the array.

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

Stability when sorting on a second key

The diagram illustrates the concept of stability in sorting. It shows three columns of data representing different sorting criteria:

- sorted by time:** Chicago 09:00:00, Phoenix 09:00:03, Houston 09:00:13, Chicago 09:00:59, Houston 09:01:10, Chicago 09:03:13, Seattle 09:10:11, Seattle 09:10:25, Phoenix 09:14:25, Chicago 09:19:32, Chicago 09:19:46, Chicago 09:21:05, Seattle 09:22:43, Seattle 09:22:54, Chicago 09:25:52, Chicago 09:35:21, Chicago 09:36:14, Phoenix 09:37:44.
- sorted by location (not stable):** Chicago 09:25:52, Chicago 09:03:13, Chicago 09:21:05, Chicago 09:19:46, Chicago 09:19:32, Chicago 09:21:05, Chicago 09:25:52, Chicago 09:35:21, Houston 09:01:10, Houston 09:00:13, Houston 09:01:10, Phoenix 09:37:44, Phoenix 09:00:03, Phoenix 09:14:25, Seattle 09:10:25, Seattle 09:22:43, Seattle 09:10:11, Seattle 09:22:54, Seattle 09:22:54, Seattle 09:36:14.
- sorted by location (stable):** Chicago 09:00:00, Chicago 09:00:59, Chicago 09:03:13, Chicago 09:19:32, Chicago 09:19:46, Chicago 09:21:05, Chicago 09:25:52, Chicago 09:35:21, Houston 09:00:13, Houston 09:01:10, Houston 09:01:10, Phoenix 09:00:03, Phoenix 09:14:25, Phoenix 09:37:44, Seattle 09:10:11, Seattle 09:10:25, Seattle 09:22:43, Seattle 09:22:54, Seattle 09:22:54, Seattle 09:36:14.

Red arrows indicate the movement of data between the first and second columns, and between the second and third columns. A red bracket labeled "no longer sorted by time" spans the second column, and another red bracket labeled "still sorted by time" spans the third column, visually demonstrating that the relative order of equal keys (like multiple entries for Chicago) is lost in the second sort but preserved in the third.

# Sorting and Stability

## Stable Sorts

- Insertion sort
- Mergesort

## Unstable Sorts

- Selection sort
- Shellsort
- Quicksort
- Heapsort

# Sorting Summary

algorithm	stable?	in place?	order of growth to sort $N$ items running time	extra space	notes
<i>selection sort</i>	no	yes	$N^2$	1	
<i>insertion sort</i>	yes	yes	between $N$ and $N^2$	1	depends on order of items
<i>shellsort</i>	no	yes	$N \log N$ ? $N^{6/5}$ ?	1	
<i>quicksort</i>	no	yes	$N \log N$	$\lg N$	probabilistic guarantee
<i>mergesort</i>	yes	no	$N \log N$	$N$	
<i>heapsort</i>	no	yes	$N \log N$	1	

Performance characteristics of sorting algorithms